Current DAD Development Plan

Jon DeVaan

May 10, 1996

Purpose

The purpose of this paper is to explain the current state of the DAD codebase's architecture and to briefly describe the strategy for moving forward.

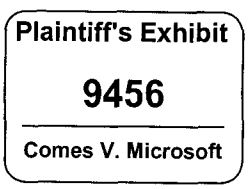
History

It is not possible to understand today's state without understanding some of the history of the products that make up DAD and Office.

Going back 10 years or more, a basic tenet of application design was to use every ounce of system resources. We even went so far as to give talks at the company meeting saying, "We want cycle sucking software." That sounds bad, but the reality was we had to shoe-horn software to fit the machines. Pretty much every decision was made based on whether the feature would fit on the floppy disk the product had to run from. Consequently, it was appropriate to think of users as single product users, and thus the notion of spending any bytes or CPU cycles on generality was discarded. Code sharing didn't make sense because it was not feasible for the user to put two products on one disk, let alone run them both at the same time. Excel, Word, and PowerPoint were all constructed during this time frame. This is important to understand because during this period, it would have been a mistake to have wasted much time attempting to create single architectures that were shared between products. There wasn't room in the executables or the users machines to pay for the size or extraneous features imposed by generality. Note that the user environment wasn't the only problem we had with any attempt at standardizing architectures. As a development organization we were not able to ship products reliably, even though the level of complexity of the products then would be laughed at by today's standards. Trying to introduce cross product requirements would have resulted in never being able to ship any products.

With the advent of Windows 3.0 and prevalent hard disks, it was possible for customers to think of using more than one program at a time. While Windows 2 allowed running more than one program at a time, no users really ever did this because most machines only had 640k of RAM and Windows didn't do that well using extended memory. Once users started running more than one program at a time, run-time integration of their programs became important to them. Windows provided cut/copy/paste and OLE 1.0 provided better abstract data type integration. Office 3.0 (Word 2.0, Excel 4.0, PP 3) was our entry product to users interested in running more than one program at a time and reasonable data integration. In terms of shared architecture, we did not advance any over the old standalone days. It is worthwhile to note that the bulk of our focus at this time was in advancing application capabilities, most notably in ease of use. It would have been possible to spend more resources on combining architectures, but we opted to advance the state of the art in the application categories. This was not an unreasonable approach as every app was embroiled in a hard competitive battle. It would have been hard to convince anyone that shared architecture should take precedence over the feature wars. The only standards we worked on were oriented around data sharing (common data formats and OLE 1.0).

Data integration was not the most important form of integration however. User interface integration proved to be far more important to our users. Office 4.0 (Word 6, Excel 5, PP 4) was our product to respond to the importance of UI integration. Office 4 represented our first attempt to complicate development processes with cross group priorities and sharing of designs. Also, the advent of OLE 2 and VBA represented the first attempt at incorporating large bodies of corporate standard technology. We could have taken this opportunity to build shared architecture for the applications. However, the bulk of sales were still via stand alone applications. The competitive battles and feature wars were also still raging. The results of the Office 4 experience are interesting. From a business point of view, it is one of the amazing success stories of all time. Internally, we were reeling from several factors. The size of the development teams required to do the quantity of work that matched our product ambitions stretched our ability to organize. The fact that we achieved UI integration by sharing designs (as opposed to sharing code) caused a huge amount of fighting between the groups requiring an amazing amount of conflict resolution. 60% of my time as development manager of Excel 5 was spent negotiating cross group problems. The shared technology we adopted from other parts of the company caused problems too. The general quality of the components was very low. Interactions with the groups responsible demonstrated a large disparity in work ethic and technical expertise. The designs for the shared technology were invasive to the incorporating applications causing performance problems. We now had a first experience with sharing technologies between groups, and the experience



STANK. The Office product was a clear success. It was clear the product groups were going to be interdependent. It was clear we had to do better.

After Office 4.x, we had arrived at a place where we had 4 mature applications built on completely different infrastructures. The penalties of the situation were clear. If we wanted to ensure UI integration we could rely on shared designs or write shared code. Shared code was not feasible because of the lack of a shared infrastructure. Shared designs were so painful to execute on that we really wanted shared feature code. The runtime size of Office was larger than users wanted. Running more than one app at a time was becoming common, and each app loaded its own version of application infrastructure. We knew we now wanted to spend the time working on Office wide architecture. However, we were still faced with strong competitors in Lotus and WordPerfect. We did not want to live in the market for the length of time required to do extensive architectural work. To overcome this we developed the "12/24" development methodology where we would place a small quantity of resources on a small release in approximately 12 months and focus the bulk of our resources on a longer 24 month cycle where we could implement some architectural changes. Thus the Office '95 and Office '97 projects were born. (Of course it was called Office '96 then, but I digress).

Office '95 was supposed to not include any new architecture or shared technology. We had formed the Office development group with the intention of focusing 100% on Office '96. However, as the plan progressed it was clear we had some opportunities. A limited amount of shared technology was developed for Office '95: File Open, FindFast, File New, File Properties, Binder, and some UI trinkets such as the Office Brand Title Bar were created. Also, some legacy code such as SDM (Standard Dialog Manager)was included. This was the first time SDM was actually shared at runtime. Previously it was linked into each application separately. We made the mistake on Office '95 of not taking how to do shared code seriously. Office '96 teams were spending a lot of time thinking through new methods for doing shared code in a higher quality more timely manner, but none of this was used on the Office '95 project. Then development experience, while better than the shared code experiences from Office 4, still was not very good.

Current Synopsis

The Office '97 project is currently working towards completion. It represents the largest re-engineering effort ever undertaken by DAD. Office '97 is amazing on a variety of levels. Over 300 developers working simultaneously. Many application infrastructure areas re-written and shared. All done in "real" time. In every dimension Office '97 is more ambitious than any other development project done in DAD, and perhaps Microsoft.

When planning Office '97 we started with the premise that we had to ship in two years, and we wanted to make headway on our goal of moving the applications onto a shared infrastructure. From the analysis, it was estimated it would take 3 major releases under the 12/24 system to complete. A diagram of the envisioned application infrastructure will be included in a future version of this memo. The decision criteria on what parts of the infrastructure to implement first represents a very organic function. We chose based on necessity of supporting shared features we wanted to implement. Many shared features represented new functionality and were relatively easy to pick. Some shared features were re-writes of existing functionality and were chosen somewhat arbitrarily, with a guess as to what we might be most able to leverage in later releases. The high level shared features are: Hyperlinks, Command Bars, Drawing, and the Office Assistant. Supporting infrastructure includes Memory management, resource management, localization support, graphics effects (gradient fills etc.), user preferences, and performance related infrastructure (delay load modules, memory grouping, etc.). Another factor in deciding what to do related to difficulty in retrofitting into the applications. If it was too hard to retrofit, we didn't do it. System data structure management like DC property caching and font handling are examples. Lastly, having to ship in two years forced us to be pragmatic.

Let's restate that pragmatic point. Some infrastructure areas were not possible to attempt because they were too contentious to arrive at a shared design. Note I am not saying they were too hard technically. We solved plenty of really hard technical problems. Let's take text as an example. There are hundreds of different text handling capabilities in our products. In the Office apps there are over five different implementations of text. Each one is tuned to a particular user scenario according to capability and performance. In reviewing the requirements I could not conceive of a way to get everyone to agree. Someone has to allow 100k growth in working set, or live without subtle bullet alignments, or without FE script features. Even with all of this we did make progress on text. The Line Services group made progress on sharing line layout. The RichEdit group created a control with enough functionality to handle a class of text users. We should take heart in that we added an application to Office, but we did not add a new text control. At least our text control per app ratio is decreasing!

Another dimension of pragmatic is the number of groups that have to cooperate in a successful solution. The higher the number, the less likely you will succeed. Text is a good example where attempting to solve too many groups' problems will doom the effort to failure.

Are we happy with what we have achieved so far? I think yes, although we certainly have a lot of problems too. The methods we chose for integrating shared code have proven to be both a blessing and a curse and will need to be rethought. The average daily quality of the shared components has not been as good as it needs to be to make sharing work well. I am not in a position to hazard a guess as to the relative improvement to Office 4.x or Office '95. Performance was a major emphasis and we have yet to see how well we'll succeed on it. Without question we have created a large amount of shared infrastructure and moved all of the Office apps on top of it.

Futures

Going forward, our stated goal is to continue filling in our infrastructure. We do not have a prioritized list of how we'll do that yet. We will follow a strategy of analyzing the gaps in the infrastructure, the methods we used to design the shared components and follow a similar prioritization scheme to Office '97.

Hopefully we can attempt sharing some of the larger more problematic infrastructure areas such as text and forms. However, there are more fundamental infrastructure areas that need to be solved first. For example, having common representations of system structures is a huge problem. Why? Because the abstractions exposed by the Win32 API are substandard for building virtually any application. Every application builds a layer over Win32. We have many of those layers, and that makes it impossible to do what we want. Take the case where multiple components draw into the same window. In order to render quickly, every component will cache GDI settings. Whoops! Component A called component B which just hosed A's GDI cache. Flush the cache! Wait! B only hoses sometimes, but I don't know when! Thus A is always slow. We need a common expression GDI cache in order to produce independent modules that can cooperate in fast redraw. Another example: Why can't we share the font dialog? Because there are over a dozen font properties that are built on top of GDI in different apps. We need a common expression of font properties if we are going to share the font dialog (let alone share a rich text data type). Once we have solved more of these nuts and bolts issues we can make better progress on higher level shared components. Another way to look at this is, any high level components we make will be substandard until we have a good low level infrastructure to share between them.

Another problematic issue looking forward is the expectation to use company standard technologies, even though the people working on the technologies don't consider application scenarios part of their problem set. Let's pick Trident as an example. Handling dialog box scenarios is not even remotely on their radar screen. Should Office use it for its dialog boxes? It will be hard to make it work at all, let alone well.

Epilogue

I have not spent any time in this memo talking about future capabilities. Mostly that is a reflection on the Office '97 emphasis on building shared infrastructure. However, I believe we will continue to do a good job on building a lot of new capabilities, similarly to the way we created DocObject, HLink and UrlMon technologies, which have since become the cornerstone of the Nashville product. It is a failing that we don't have a better list of directions on what new capabilities we are thinking about.

I have not spent any time talking about the huge amount of architectural and capability work that gets done in each application team. Examples are the new "perfect" dependency scheme in Excel '97, or background grammar in Word '97. Other investments are the single pass save in Word, change logs and multiple undo in Excel, and multi-user editing. A shortcoming of DAD in general has been the ability to roll up the entire list of innovations so we can present it in one sitting.

I did not address new user paradigms or application models as that is the subject of a different write-up.